Aggressive Snoop Reduction for Synchronized Producer-Consumer Communication in Energy-Efficient Embedded Multi-Processors

Chenjie Yu
University of Maryland
College Park, USA
purety@umd.edu

Peter Petrov
University of Maryland
College Park, USA
ppetrov@ece.umd.edu

ABSTRACT

Snoop-based cache coherence protocols are typically used when multiple processor cores share memory through a common bus. It is well known, however, that these coherence protocols introduce an excessive power overhead. To help alleviate this problem, we propose an application-driven customization technique where application knowledge regarding data sharing in producer-consumer relationships is used in order to aggressively eliminate unnecessary and predictable snoop-induced cache tag lookups even for references to shared data, thus, achieving significant power reduction with minimal hardware cost. Snoop-induced cache tag lookups for accesses to both shared and private data are eliminated when it is ensured that such lookups will not result in extra knowledge regarding the cache state in respect to the other caches and memories. The proposed methodology relies on the combined support from the compiler, the operating system, and the hardware architecture. Our experiments show average power reductions of more than 80% compared to a general-purpose snoop protocol.

Categories and Subject Descriptors: B.3 [Hardware]: Memory structures; C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems

General Terms: Algorithms, Design, Experimentation

1. INTRODUCTION

The abundance of wireless connectivity coupled with the ever growing increase in integration densities have resulted in a multitude of hand-held and wearable embedded applications such as MP3 players, mobile phones with aggregate data functions, personal organizers, etc. Battery life and power consumption has become one of the primary implementation constraints for these applications. Low-power consumption is of great importance to stationary devices as well, such as game consoles and set-top boxes, which are directly connected to the power grid. Excessive power consumption for these devices requires costly and bulky heat dissipation technologies.

Due to the integration of multiple functionalities and ever increasing demand for performance, it has become a natural design practice to utilize *Multi-Processor Systems-On-a-Chip (MPSoC)* in embedded systems. Typically these systems feature several processor cores, possibly of heterogeneous nature, that access a shared memory. The accesses to the shared memory can be implemented with various interconnect topologies. However, for reasons of low complexity and high speed, the most common approach is to use a common system bus. In order to provide the required bandwidth to the shared memory, local caching at each processor is usually employed. Local caching, however, introduces the possibility of cache incoherence when a processor updates a data after that same data is cached somewhere else. To resolve this issue, cache coherence protocols must be implemented in such systems. The snoop-based cache coherence protocols are the most widely deployed as they rely on the inherent broadcast nature of the common bus. Each cache controller "snoops" the bus for memory transfers, for each of which a cache lookup is performed in order to determine whether a cache line state should be changed in the local cache. However, this protocol tends to be overly conservative in many real world programs, especially embedded applications.

Previous research [1] has shown that only around 10% of the application memory references actually require cache coherence tracking. Quite often data are cached in just a few nodes and snooping in the others leads to waste of energy. It has been reported that snoop-related cache activities can contribute for up to 40% of the total cache power [2, 3].

To address this problem, we introduce a methodology which aggressively eliminates the majority of snoop-induced cache lookups and thus, achieves significant power reduction. The proposed technique explicitly exploits application-specific information regarding the exact *producer-consumer relationships* between various tasks as well as information regarding *the precise timing of synchronized accesses* to shared memory buffers by their corresponding producers and/or consumers. This program knowledge is used to eliminate a large number of snoop-induced cache lookups *even for references to the shared memory buffers*. In contrast to general-purpose programs, such application knowledge is easily available in embedded design environment, where system designers have full knowledge and control in coordinating and fine-tuning the interaction for each specific programs between hardware and operating system.

The proposed methodology actively eliminates the majority of unnecessary snoop-induced cache tag look-ups. The conventional snoop controllers are augmented with small additional hardware which is controlled by the operating system. This hardware dynamically identifies accesses to relevant memory regions and precludes snooping to those regions whenever it is safe to do so, thus saving a significant amount of power.

2. RELATED WORK

The emergence of multi-core processors in embedded applications has exacerbated the power concerns with these applications

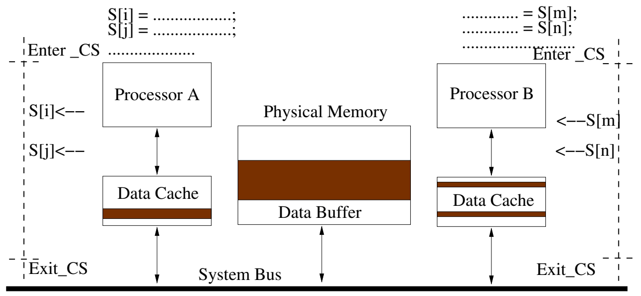

Figure 1: P/C communication with shared memory

but also has exposed more opportunities to reduce power consumption. Many research projects have tried to address power-aware coherence protocol problems. However, most of them are targeting general purpose systems. Few of them have been in the embedded system domain, which often times exhibit specific hardware architectures and program behaviors and usually expose more stringent power constraints. This is where our major contributions would go.

Jetty [4] is the name of a family of snoop filters designed to reduce energy consumption in snoopy bus-based multiprocessor systems. By observing and recording recent cache behaviors in additional hardware structures, Jetty can provide information as "what is present in the local cache and what is not". By feeding this information to snooping controllers, certain snooping can be avoided at relatively low cost. The authors report an average of 29% energy reduction for L2 caches. RegionScout [5] is another family of snoop filters that exploit coarse-grain data sharing information to reduce energy and bus traffic. RegionScout is also designed for server applications. In RegionScout, memory is divided into a number of regions. The hardware keeps records as which region contains shared content in the local cache so as to preclude lots of unnecessary snooping. In [6] the authors have introduced a technique for run-time identification of data streams. A large number of coherence misses are eliminated by dynamically identifying sequences of memory accesses which correspond to a data stream. Coherence misses are eliminated by moving the data stream to the requesting processor in advance.

While techniques for general purpose applications report significant power benefits, their hardware overhead would be non-trivial in most embedded systems. Moreover, in embedded applications, system designers usually have much more detailed control over programs and hardware so that data sharing information is precise and deterministic. By exploiting this advantage, we can avoid the speculative mechanisms present in general-purpose approaches and minimize area and power overhead. Furthermore, since we are exploiting producer-consumer relationships in typical applications, much stronger run-time sharing patterns can thus be exploited, yielding even greater energy reductions.

3. FUNCTIONAL OVERVIEW

The proposed technique benefits from the availability of precise application information regarding producer/consumer relationships in many embedded applications. The producer-consumer relationship between different processing nodes occurs naturally in the presence of data sharing. Quite often, sharing exists only within a small number of nodes rather than across the entire system. In this case, data would not be cached in other nodes' caches and probing those caches for shared data is unnecessary. Furthermore, shared data buffers are essentially temporally "private" when access right to them is acquired by a certain node. Consequently, for a certain period of time even snooping for those shared data may be eliminated. Thus, by precisely differentiating different memory regions that store shared data and the exact relationships between them, we can enforce a more energy efficient coherence protocol implementation, which is active only during the ownership transition of shared data blocks.

3.1 Synchronized P/C Communication

In many multi-tasking embedded applications, especially stream applications, communication between different processing nodes constitutes Producer-Consumer (P/C) relationships, as illustrated in Figure 1. In this example, processor A performs a computation on batches of input data and for each batch it stores its output to the shared data buffer S. Data stored in this buffer are read by the task running on processor B performing subsequent computations. With respect to data buffer S, processor A is producer, while processor B servers as consumer. Processor B, in turn can serve as producer to other shared memory buffers, which are "consumed" by other processors (possibly processor A). Since processors often read and write from different buffers during a program run, they may well be producers and consumers at the same time, with respect to different shared buffers. In order to prevent non-deterministic behavior and race conditions, accesses to the shared buffers must be fully synchronized. To obtain exclusive access to the shared buffer S, both producer and consumer use synchronization primitives such as locks, semaphores, or barriers. The portions of the code where the shared buffer is worked on are typically referred to as *Critical Sections (CS)* and are surrounded with synchronization operations in the forms of *ENTER_CS* and *LEAVE_CS*.

In the example in Figure 1, processor A is writing to buffer S, which results in invalidations in all others' caches, including the cache of processor B, and triggers snoop-induced cache lookups. Similarly, when processor B acquires access to buffer S and starts consuming, it generates a large number of read-misses, which in turn triggers snoop activities at all remote caches including the one of processor A. Most of these snooping and cache tag lookups are unnecessary and as such are eliminated by our approach.

3.2 Snoop-Phases in P/C Communication

The snoop activities at both producer and consumer follow a certain well-defined pattern that constitutes of two phases.

Phase one occurs in the beginning of a critical section. As is shown, processor B's cache contains blocks belonging to the shared buffer S. These cache blocks have been brought into B's cache while it was "consuming" S in the previous critical section. Since processor A is writing to S, these blocks will be invalidated before the end of the critical section. Time moment t_1, shown in Figure 2a, corresponds to such a state. Before these cache blocks are entirely invalidated, snoop protocol should work to ensure coherence.

As processor A's computation goes on, the number of cache lines containing S at B monotonically decreases. The rate of this decrease varies with applications behaviors and hardware configurations. At some point, the number of cache lines at B holding S drops to zero. Apparently, from this point on, no snooping for S is needed at processor B, because B's cache contains no data of S that may cause coherence problems. In the proposed methodology, we refer to this moment as *Snoop-Phase Transitioning (SPoT) Point*.

The *second snoop phase* starts from SPoT point and lasts until the end of the critical section; with respect to the snoop operations for the shared buffer, phase two effectively lasts until the beginning of the next data consumption iteration (critical section). Time instance t_2, as shown in Figure 2b, has occurred during the second phase when B's cache is free of any content of S. It is evident that during the second phase B does not have to check its cache for such misses, since it is holding no valid lines belonging to that

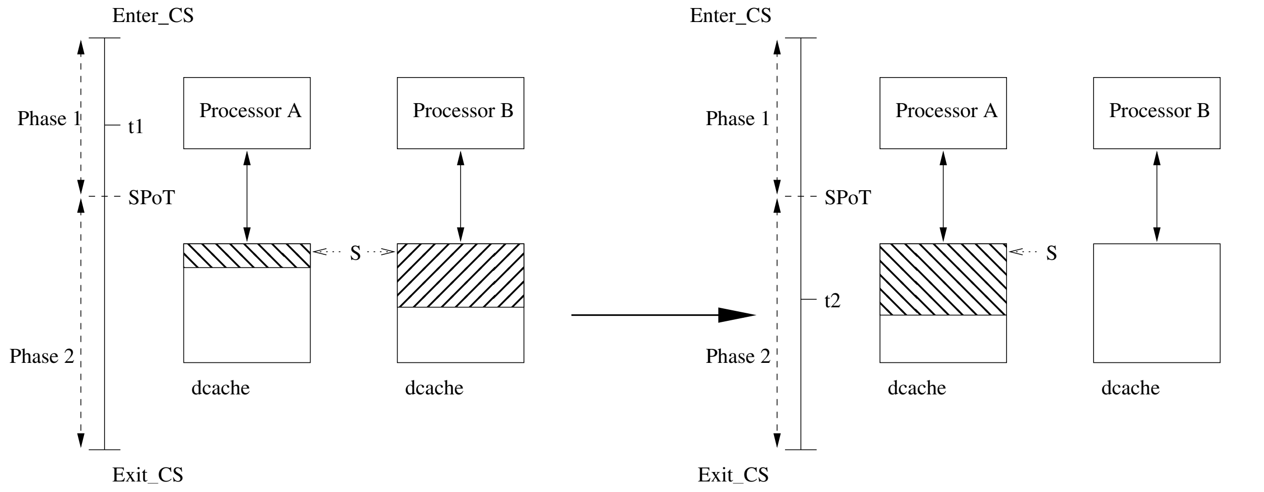

Figure 2: P/C cache snooping activities

region and no new lines from S will be brought in until the next consumption iteration.

As can be seen from these observations, the potential savings from disabling snooping at B for references to S depend on how soon the SPoT point occurs in the production cycle of A. If SPoT comes near the end of the critical section, the benefits would be limited. However, our experiments on a set of benchmarks show that SPoT occurs quite fast. First, while A is producing to S, B is most likely doing other useful computations other than idling and thus will touch other shared data buffers as well as private data. These activities would thrash B's cache and evict S quickly. Second, A could be touching data from buffer S in some irregular fashion that would expedite invalidation in B's cache. For example, matrix multiplication usually touches different cache lines much faster than striding access. Moreover, if buffer S is relatively big compared to cache size, many blocks that contain S may have already been evicted at the very beginning at the critical section.

The situation when B is consuming from S and A is performing other computations is similar. The difference is that, the number of modified, or "dirty", cache lines from S at A would determine the SPoT point. Following the same arguments as above, one can notice that the number of "dirty" cache lines from S at A monotonically decreases and that it often reaches zero quickly. Consequently, snoop-induced cache lookups at A for references to S (generated at B) would be redundant after this SPoT, and can be safely eliminated.

3.3 Snoop-Phase Detection

By exploiting P/C relationship and identifying SPoT for each shared memory buffer, the snoop controllers can aggressively filter subsequent snooping to these regions in their respective caches, and thus, achieve significant energy reduction. Snoop-induced cache lookups at each processor will be allowed only for small subsets of memory references to shared buffers of that processor, which occur only before SPoT points of associated regions. All other snoops to the shared region after SPoT are safely removed without violating coherence between different caches.

In order to exploit the two snoop-phase pattern and, thus, filter snooping for accesses to particular shared buffers, a mechanism is needed to distinguish between references to various shared buffers in multiprocessor systems. The approach we propose here relies on the help of the operating system memory manager to assign such unique identifiers. The workings of this identification scheme are outlined in the next subsection.

```
                                int S[100];
                                ..
                                ..
Thread 1                        Thread 2

..                              ..
RegisterRegion(S, 100, producer);   RegisterRegion(S, 100, consumer);
..                              ..
..                              ..
ENTER_CS(S);                    ENTER_CS(S);
...                             ...
...                             ...
LEAVE_CS(S);                    LEAVE_CS(S);
```

Figure 3: Transferring to OS shared buffers information

We propose two methodologies for snoop elimination based on the snoop-phases for P/C communication:

Passive SPoT Detection. This approach is based on direct observation of SPoT during run-time. The implementation involves several hardware counters which monitor the data sharing status with respect to buffer IDs associated to cache lines. These counters keep track of the number of valid cache lines (for consumer tasks) or dirty cache lines (for producer tasks) belonging to different shared buffers.

Active SPoT Migration. In the second approach, which is an optimization over the passive SPoT detection, a special action is taken at each processor when exiting a critical section to make sure that the relevant cache lines are either invalidated (consumer) or changed to shared state (producer). This is achieved through the use of a simple hardware mechanism activated when a task exits its critical section. In effect, this ensures that the SPoT points occur earlier as compared to their natural timing.

3.4 Shared Buffer Identification

The proposed approach distinguishes the shared buffers by letting each task inform the operating system (through a simple API) as of which shared buffers are used in that task, where is the critical section for each of the buffers, and whether the buffer is being accessed in a producer or a consumer matter. The OS memory manager subsequently assigns an unique identifier for each such shared buffer and tags all the memory pages belonging to that buffer with this identifier. The buffer identifier associated to each page is captured in the MMU and the page table within the OS. For each memory reference that is placed on the bus, the buffer identifier is obtained from the MMU (together with the physical address of the location) and placed on the bus as part of the memory transaction.

Figure 3 illustrates how the information regarding shared memory buffers and the relationship of task to buffers (producer or consumer) is transfered to the OS. Often times multitasked applications are developed by using multithreading libraries. Threads are created, terminated, and synchronized at the application level without intervention from the kernel, thus achieving high efficiency. Because the multiple threads, comprising the application, execute within the same address space (they share a single OS-level process), it is impossible for the OS to determine which memory buffers from that shared address space are actually shared between the threads. This information, however, can be easily provided by the software developer by using a special function, which interfaces with the OS. Following this approach during the thread initialization phase, this function will be called to register all the shared buffers associated with each thread as shown in Figure 3.

The OS subsequently assigns a unique buffer identifier for each shared buffer. As the identification occurs at page granularity level, shared buffers need to be aligned at page boundaries, which is easily done by the compiler. Our experiments show that supporting up to 16 different shared buffers in the system would be enough for many multitasked applications. Thus, a 4-bit *Buffer Identifiers (BID)* would be assigned to each memory page and captured in the TLB. In the subsequent section we outline how BIDs are used and explain (and later demonstrate by experimental results) that their overhead is very small.

4. PASSIVE SPOT DETECTION

The passive SPoT detection approach relies on hardware to detect SPoT occurrences and block snoop lookups for references to shared buffers. The hardware architecture is shown in Figure4.

As can be seen, an additional set of BIDs are attached to the TLB. These shared buffer IDs are also associated with the page table entries and maintained by the operating system. When page table entries are loaded into the TLB, their associated BIDs are also fetched. The small table of BIDs associated to the TLB is implemented as a separate SRAM array and is not read and compared as a part of the normal TLB lookup which is needed for cache access. A similar small SRAM array is implemented in order to associate BIDs with each cache line. When a new cache line is brought into the cache, its BID is stored in that array. Buffer IDs are never compared like cache tags or even read when the cache line is accessed. They are only used as indexes to a small set of counters. In our experiments we have modeled 4-bit BIDs which require 16 counters. This is enough to handle a total of 14 shared buffers in the system - the remaining two values are used to mark private data pages which do not need snooping at all and memory pages whose sharing pattern is "unknown", e.g. OS data, and always need snooping to ensure coherence. Because of the very small bit-width of the BIDs, the area overhead of these small BID SRAM arrays is minimal compared to other components in the cache system.

The BIDs associated with cache lines are used in the process of SPoT detection. The hardware counters that BIDs index to are used to keep track of the number of cache lines that hold data belonging to shared buffers. An additional Producer/Consumer (P/C) bit indicates what type of cache lines must the counters track according to the role of the tasks as a producer or a consumer. This bit is also assigned by the operating system at task initialization.

The counters act differently according to the producer/consumer bit. For producer tasks they keep track of the number of dirty cache lines of the buffers. The producer (or "dirty") counter associated to a buffer increments on a write miss and a new line is brought into the cache, or when there is a write hit on a "read-only" line that would turn to "dirty" or "modified" as both actions introduce a new dirty cache line into that buffer. The counter decrements when a dirty line is evicted or when there is a read miss from other processor and a dirty line is changed from "modified" to "read-only/shared". Similarly, the consumer task counters keep track of the number of valid cache lines that belong to the particular shared buffers. They increment when a new cache line is brought in on a read miss and decrement whenever a valid line is evicted or invalidated. All the counters are set to 0 at the beginning of execution.

When the value of a counter is zero, there are no cache lines corresponding to the associated shared buffers in the local cache and snooping can be blocked for them. Such blocking is achieved through a *Snoop Blocking Register (SBR)*. This is a simple bit-mask register with a bit per shared buffer, where the bits are directly indexed by the buffer ID; a zero at bit position indicates that snoop-induced cache lookups for this buffer are blocked. When the counters reach zero, the corresponding bit in the SBR is set to 0.

When a memory request in the form of a read-miss or a write-miss is place on the bus, the BID of the address is obtained from the BID table and is placed on the bus as a part of the transaction. Note that no additional bus lines are needed as the data lines for such transactions are used to carry BIDs. The cache controllers snoop on the bus as usual, only that the SBR register is checked prior to performing the a cache-lookup. References to private regions (BID=0000) are always blocked, while references to "unknown" regions (BID=1111) are always snoop-enabled.

Overhead Analysis. The area overhead is dominated by the two small SRAM arrays that associate BIDs to TLB entries and individual cache lines. Compared to the cache and TLB sizes, this area overhead is typically below 5%. The power overhead is comprised of reading the BIDs and incrementing/ decrementing the buffer counters. Additionally, on a cache miss the BID is read and placed on the bus. All these events only occur on cache misses or certain cache line state changes and as such are not significant as compared to the savings achieved. In our experiment results we have accounted for all these overheads, including the power needed to transfer on the bus the 4-bit BIDs.

5. ACTIVE SPOT MIGRATION

Clearly, the earlier in the critical section SPoT occurs, the better the savings achieved by the proposed methodology. The SPoT can be actively moved earlier in time by making sure that the local cache is released from cache lines that hold content of shared buffers. A small additional hardware can be employed for this purpose to forcefully expedite the occurrence of SPoT by either writing back dirty lines (and thus changing them to "shared/read-only") for producer tasks, or by invalidating valid lines for consumers. For this purpose, we have experimented with a simple hardware mechanism which simply traverses the BID array associated with the cache lines and either write-backs (for a producer buffer) or invalidates (for a consumer) the cache lines if the BIDs match. This procedure need to be initiated at the exit from the critical section. BIDs can be checked in parallel as their width is rather small.

On the producer side, each buffer ID is checked if it matches the BID of the current critical section and, if positive, the state of the cache line is changed and the data is written to memory. This necessitates checking the Valid and the Dirty bits of the cache line only, thus minimizing the overhead. The procedure continues until the associated counter reaches zero. At that moment snooping for that buffer can be safely blocked until the next cycle of production. The duration of this process depends on how fast the memory subsystem supports for cache write backs. However, no extra power is being added as these cache lines would have to be written to memory anyways, though maybe later in the execution cycle when the consumer requests them.

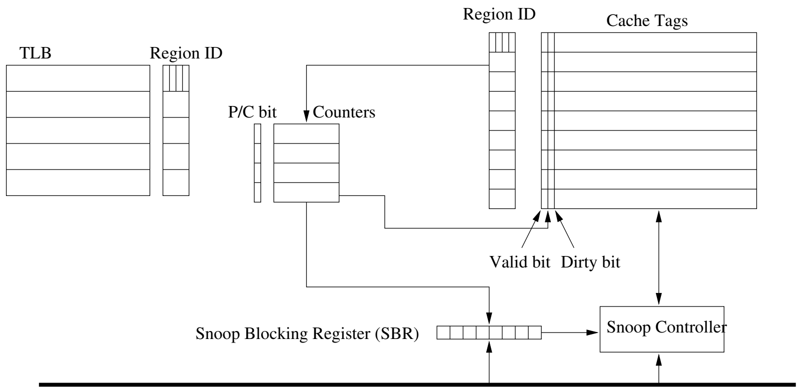

Figure 4: Hardware architecture for SPoT detection

Similar steps are needed at the consumer's side - in this case, however, lines from the buffer need only to be invalidated if present, which only means clearing the valid bit of the matching cache lines.

In our experiments we have noticed, however, that often times only a few dirty/valid lines are left in the cache after exiting the critical section, due to other memory activities. Consequently, the expected number of write-backs is relatively small. In our experiments, we have used a fixed number of cycles which assumes the worst case when all the cache lines are dirty (or valid) for the shared buffers. Even for this very conservative assumption, the achieved results are still quite positive.

6. EXPERIMENTAL RESULTS

We have conducted detailed simulations on a set of multitasked applications. The systems have four processors, each performing one computational task on a number of shared data buffers. The four processors are producers and consumer to one another. Except for the very first one all are both producers and consumers and work on different data buffers in different synchronization sections. The individual tasks constitute of: *FFT*, *ADPCM*, *matrix multiplication*, *data encryption tasks*, *lzo-compression*, *g721*, image processing - the *blur* and the *edge-detection*, and *video processing*. The tasks cover benchmarks from the Mediabench [7] and MiBench [8] suits, as well as from other open-source image and video processing tools. The multitasking applications we have used are: *A1*={*LU, MMUL, rijndael, lzo*}; *A2*={*FFT, g721, blowfish, sha*}; *A3*={*blur, edge-detection, rijndael,lzo*}; *A4*={*FFT, fdct, IFFT, AES*}, which represent embedded applications in digital filtering, audio, image, video processing and security areas.

We have used the M5 [9] simulator to perform our experiments. The simulator is used in system-call emulator mode and is extended with a collection of thread synchronization primitives. The simulated hardware configuration is of four processors (executing the Alphs ISA) connected to a shared memory through a common bus. We have experimented with four cache organizations: caches of size 16kB and 32kB, either direct-mapped or 4-way set-associative. The data buffers used for communication between the processor nodes are of sizes 16kB and 64kB; experiments for both data sizes have been conducted.

The cache power expenditure of the four cache configurations have been obtained through Cacti v4.2 tool [10] for 0.18μm technology. The energy associated with the additional hardware structures for the proposed methodology are evaluated as follows: The buffer IDs are modeled as small SRAM tables, whose energy is similarly obtained by Cacti. The SBR is accessed on every bus transaction and is implemented as a 16-bit register. The counters' actions are accounted for on a cache line replacement or state

	16k4W	16kDM	32k4W	32kDM
A1	37/1,241	88/2,349	58/1,001	85/1,172
A2	11/427	12/2,594	24/180	25/2,348
A3	47/1,060	58/1,127	46/846	66/972
A4	23/5967	18/604	24/247	31/423

Table 1: Number of snoops (x1000) for 16kB data buffers

	16k4W	16kDM	32k4W	32kDM
A1	14/18,590	64/22,308	28/8,935	138/8,371
A2	45/3,953	54/23,960	69/3,819	61/23,304
A3	105/2,435	133/2,643	153/2,231	185/2,398
A4	88/7,124	177/30,989	92/6,864	177/30,683

Table 2: Number of snoops (x1000) for 64kB data buffers

change; we have modeled them as 12-bit up-down counters. We have also taken into account the overhead of placing the 4-bit BIDs along with addresses on the bus. The energy data reported in [11] has been scaled to 4 bus lines with 50% bit-transition activities.

We have chosen the most widely used snoop protocol with four symmetrical processors connected to a common bus as baseline. The snooping activities for the baseline and the proposed Passive-SPoT detection approach are shown in Table 1 and Table 2. The numbers in the tables come in pairs, the first showing the total number of snooping (x1000) achieved by the Passive-SPoT technique, while the second number being the baseline snoop activities.

It is clear from the results that the Passive-SPoT-Detection methodology significantly reduces the amount of snoop-induced cache lookups. The achieved reductions vary with different shared buffer sizes and cache sizes and organizations and the nature of different kernels. In general, the larger the cache size is, the smaller the snoop reductions of the Passive-SPoT-Detection. This is because caches with bigger capacity can hold more cache lines and hold them longer and this defers the occurrence of SPoT. However, larger caches also mean bigger area and higher power consumption. Increased associativity has a small impact on the reductions. However, since more cache tags are to be checked in parallel on every access to the cache, the energy saving is even more significant with our technology. Consequently, the actual energy reductions for larger size and higher associativity caches would be more significant than for those with smaller sizes and associativities. The actual energy numbers, after taking into account the introduced hardware and power overheads, are shown in Table 3 and Table 4. Similarly, the numbers come in pairs Passive-SPoT vs. baseline.

Figure 5 and Figure 6 show the achieved energy reductions in percentage and compare them across the four cache organizations. These figures also show the energy reductions achieved by the Active-

	16k4W	16kDM	32k4W	32kDM
A1	8/74	13/50	8/67	9/48
A2	3/25	13/81	2/12	12/96
A3	8/63	7/35	7/57	7/40
A4	4/36	3/19	3/17	3/17

Table 3: Energy consumption (μJ) for 16kB data buffers

	16k4W	16kDM	32k4W	32kDM
A1	89/1,109	108/473	44/601	45/341
A2	21/236	115/745	22/257	113/949
A3	17/145	167/82	20/150	18/98
A4	39/425	152/964	38/462	152/1,249

Table 4: Energy consumption (μJ) for 64kB data buffers

SPoT Migration methodology. Each hardware configuration is represented by a pair of bars. The first bar represents the Passive-SPoT, while the second one corresponds to the Active-SPoT methodology. It is evident that the proposed techniques significantly reduce snoop power for the cache system. The average is above 80%, after taking into account the introduced overhead. As can be seen from the figures, the active approach produces slightly better results than the passive one. This is largely due to our evaluation methodology which assumes the worst case number of cycles needed for traversing all the BIDs after leaving the critical sections. It is clear that in reality much smaller number of cache lines would need to be checked in the cache which should result in faster SPoT occurrence. Moreover, since the reductions achieved by the Passive-SPoT approach are already significant, there is limited space for the Active-SPoT to eliminate more lookups.

7. CONCLUSIONS

In this paper, we have presented a methodology for snoop power reduction in embedded multiprocessors. Through the cooperation of software developer, compiler, operating system, and hardware architecture, a fine-grained application knowledge regarding *producer-consumer relationships* between tasks and *precise timings of the synchronized accesses* to shared buffers is exploited to aggressively eliminate a large number of snoop-induced cache lookups even for references to shared buffers. The introduced hardware is not only cost-efficient but is also software-programmable and can capture and utilize the detailed and deterministic application sharing information. The proposed methodology is ISA-independent and, thus, can be applied to any modern or future energy-efficient homogeneous or heterogeneous embedded multiprocessor platform.

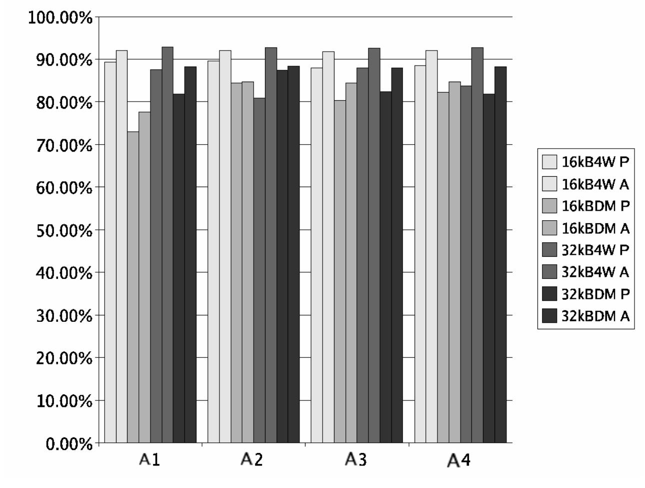

Figure 5: Energy reduction for 16kB shared data buffers

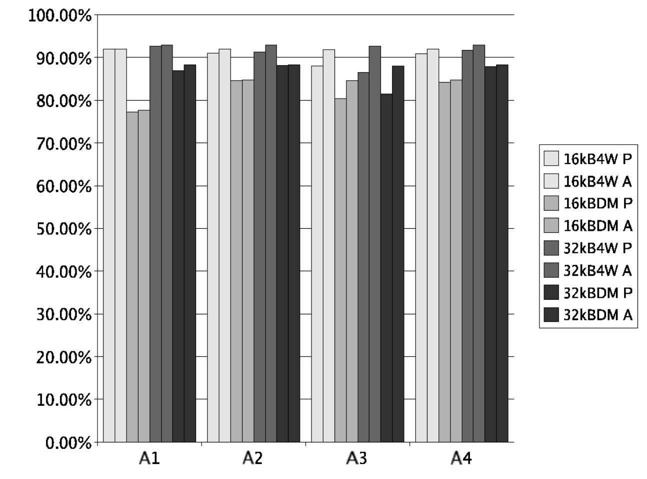

Figure 6: Energy reduction for 64kB shared data buffers

Acknowledgement: We want to thank Dr. Donald Yeung's group for making available to us their M5 simulator facilities and their assistance in using it.

8. REFERENCES

[1] Jim Nilsson, Anders Landin and Per Stenstrom, "The Coherence Predictor Cache: A Resource-Efficient and Accurate Coherence Prediction Infrastructure", in *ISPDP*, 2003.

[2] M. Ekman, F. Dahlgren and P. Stenstrom, "TLB and snoop energy-reduction using virtual caches in low-power chip-microprocessors", in *ISLPED*, pp. 243–246, August 2002.

[3] M. Loghi, M. Letis, L. Benini and M. Poncino, "Exploring the energy efficiency of cache coherence protocols in single-chip multi-processors", in *GLSVLSI*, pp. 276–281, 2005.

[4] A. Moshovos, G. Memik, A. Choudhary and B. Falsafi, "JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers", in *HPCA*, 2001.

[5] A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence", in *ISCA*, 2005.

[6] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki and B. Falsafi, "Temporal Streaming of Shared Memory", in *ISCA*, 2005.

[7] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", in *30th MICRO*, pp. 330–335, December 1997.

[8] M.R Guthaus, J. S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", in *WWC*, pp. 3–14, Dec 2001.

[9] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi and S. Reinhardt, "The M5 Simulator: Modeling Networked Systems", *IEEE Micro*, vol. 26, n. 4, pp. 52–60, 2006.

[10] D. Tarjan, S. Thoziyoor and N. Jouppi, "CACTI 4.0: An Integrated Cache Timing, Power and Area Model", Technical report, HP Laboratories Palo Alto, June 2006.

[11] R. Bashirullah, W. Liu and R. Cavin, "Low-power design methodology for an on-chip bus with adaptive bandwidth capability", in *DAC*, pp. 628–633, 2003.